CS64 Week 5 Lecture 1

Kyle Dewey

Overview

- More branches in MIPS
- Memory in MIPS
- MIPS Calling Convention

More Branches in MIPS

- `else_if.asm`
- `nested_if.asm`
- `nested_else_if.asm`

Memory in MIPS

Accessing Memory

- Two base instructions: load-word (`lw`) and store-word (`sw`)
- MIPS lacks instructions that do more with memory than access it (e.g., retrieve something from memory and add)
    - Mark of RISC architecture

Global Variables

- Typically, global variables are placed directly in memory, not registers
- Why might this be?

Global Variables

- Typically, global variables are placed directly in memory, not registers
- Why might this be?
  - Not enough registers

Global Variable Example

- `access_global.asm`

Arrays

- Question: as far as memory is concerned, what is the major difference between an array and a global variable?

Arrays

- Question: as far as memory is concerned, what is the major difference between an array and a global variable?
    - Arrays contain multiple elements

Array Examples

- `print_array1.asm`
- `print_array2.asm`
- `print_array3.asm`

-print_array1.asm: typical index-based loop
-print_array2.asm: slightly optimized form of print_array1.asm
-print_array3.asm: pointer arithmetic based loop, with fewer instructions

MIPS Calling Convention

Functions

- Up until this point, we have not discussed functions
- Why not?

Functions

- Up until this point, we have not discussed functions
- Why not?
  - Memory is a must for the call stack
  - ...though we can make some progress without it

-Because of things like recursion, we generally don't even know ahead of time how many variables we are going to need. This is what we have the stack for.

Implementing Functions

- What capabilities do we need for functions?

Implementing Functions

- What capabilities do we need for functions?
  - Ability to execute code elsewhere
  - Way to pass arguments
  - Way to return values

Implementing Functions

- What capabilities do we need for functions?
    - Ability to execute code elsewhere - branches and jumps
    - Way to pass arguments - registers
    - Way to return values - registers

Jumping to Code

- We have ways to jump to code
- What about jumping back?

```
void foo() {
   bar();
   baz();
}
```

```
void bar() {
   ...
}
```

```
void baz() {
   ...
}
```

Jumping to Code

- We have ways to jump to code
- What about jumping back?
    - Need a way to save where we were
    - What might this entail on MIPS?

```
void foo() {
  bar();
  baz();
}
```

```
void bar() {
  ...
}
```

```
void baz() {
  ...
}
```

Jumping to Code

- We have ways to jump to code
- What about jumping back?
    - Need a way to save where we were
    - What might this entail on MIPS?
        - A way to store the program counter

```
void foo() {
  bar();
  baz();
}
```

```
void bar() {
  ...
}
```

```
void baz() {
  ...
}
```

Calling Functions on MIPS

- Two crucial instructions: `jal` and `jr`
- `jal` (jump-and-link) will simultaneously jump to an address, and store the location of the **next** instruction in register `$ra`
- `jr` (jump-register) will jump to the address stored in a register, often `$ra`

Calling Functions on MIPS

- `simple_call.asm`

Passing and Returning Values

- We want to be able to call arbitrary functions without knowing the implementation details
- How might we achieve this?

Passing and Returning Values

- We want to be able to call arbitrary functions without knowing the implementation details
- How might we achieve this?
    - Designate specific registers for arguments and return values

Passing and Returning Values on MIPS

- Registers `$a0 - $a3`: argument registers, for passing function arguments
- Registers `$v0, $v1`: return registers, for passing return values

Passing and Returning Values on MIPS

- `print_ints.asm`
- `add_ints.asm`

Problem

- What about this code makes this setup break?

```
void foo() {
  bar();
}
void bar() {
  baz();
}
void baz() {}
```

Problem

- What about this code makes this setup break?
  - Need multiple copies of `$ra`

```
void foo() {
  bar();
}
void bar() {
  baz();
}
void baz() {}
```

-We'd have to copy the value of $ra to another register before calling another function
-This can be done, but eventually we're going to run out of registers. Call stacks more than 32 functions deep are common in practice, so we can't possibly store everything in registers

Another Problem

- What about this code makes this setup break?

```
void foo() {
  int a0, a1, ..., a20;
  bar();
}
void bar() {
  int a21, a22, ..., a40;
}
```

Another Problem

- What about this code makes this setup break?
  - Can't fit all variables in registers at the same time. How do I know which registers are even usable without looking at the code?

```
void foo() {
  int a0, a1, ..., a20;
  bar();
}
void bar() {
  int a21, a22, ..., a40;
}
```

-With knowing which registers are usable, both foo and bar can't use, say, $t0 at the same time, or else they might step on each other's toes. If foo sets $t0 to some value and then calls bar, if bar sets $t0 to some other value, then this might mess things up in foo when bar returns if foo still needs the value in $t0

Solution

- Store certain information in memory at certain times
- Ultimately, this is where the call stack comes from

Who saves what?

- Certain registers are designated to be preserved across a call
    - Preserved registers are saved by the function called (e.g., `$s0 - $s7`)
    - Non-preserved registers are saved by the caller of the function (e.g., `$t0 - $t9`)
- Question: why a split?

Who saves what?

- Certain registers are designated to be preserved across a call
    - Preserved registers are saved by the function called (e.g., `$s0 - $s7`)
    - Non-preserved registers are saved by the caller of the function (e.g., `$t0 - $t9`)
- Question: why a split? - not everything is worth saving

Saved where?

- Register values are saved on the stack
- The top of the stack is held in `$sp` (stack-pointer)
- The stack grows from high addresses to low addresses

Register Saving Example

- `save_registers.asm`

Recursion

- This same setup handles nested function calls and recursion - we can save `$ra` on the stack
- **Example:** `recursive_fibonacci.asm`